Path-based and triplification approaches to mapping data into RDF: user behaviours and recommendations

Paul Warren[a,1] , Paul Mulholland[a], Enrico Daga[a], Luigi Asprino[b]
[a]*Knowledge Media Institute, The Open University, Milton Keynes, MK7 6AA, U.K.*
[b]*University of Bologna, Via Zamboni 32, Bologna, Italy*

Abstract. Mapping complex structured data to RDF, e.g. for the creation of linked data, requires a clear understanding of the data, but also a clear understanding of the paradigm used by the mapping tool. We illustrate this with an empirical study comparing two different mapping paradigms from the perspective of usability, in particular from the perspective of user errors. One paradigm uses path descriptions, e.g. JSONPath or XPath, to access data elements; the other uses a default triplification which can be queried, e.g. with SPARQL. As an example of the former, the study used *YARRRML*, to map from CSV, JSON and XML to RDF. As an example of the latter, the study used an extension of SPARQL, *SPARQL Anything*, to query the same data and CONSTRUCT a set of triples. Our study was a qualitative one, based on observing the kinds of errors made by participants using the two paradigms with identical mapping tasks, and using a grounded approach to categorize these errors. Whilst there are difficulties common to the two paradigms, there are also difficulties specific to each paradigm. For each paradigm, we present recommendations which help ensure that the mapping code is consistent with the data and the desired RDF. We propose future developments to reduce the difficulty users experience with YARRRML and SPARQL Anything. We also make some general recommendations about the future development of mapping tools and techniques. Finally, we propose some research questions for future investigation.

Keywords: Mapping to RDF, YARRRML, SPARQL Anything, Usability

1. Introduction

Structured data exists in a wide variety of formats, e.g. CSV, JSON, XML, and HTML. The need to convert this to RDF, for the creation of linked data, has stimulated the development of a variety of mapping techniques. There are two broad paradigms. One uses path descriptions, e.g. with JSONPath or XPath, to identify data elements. The other uses a default triplification which can be queried, e.g. with SPARQL.

An example of the use of path descriptions is R2RML [1], which was developed to map from relational database format to RDF. This was extended to RML [2], which also maps from CSV, TSV, JSON and XML. For the latter two formats, RML makes use of JSONPath and XPath. Subsequently, YARRRML, [4], has been developed as a more human-friendly representation of R2RML and RML rules. Rules written in YARRRML are translated to RML, which is then used to map the source data to RDF.

An example of the use of triplification is SPARQL Anything [5], which enables SPARQL to be used with structured data, e.g. CSV, JSON, XML, HTML, TXT and Markdown. The SERVICE operator is used to identify the relevant document and a default triplification of the whole document is created automatically, the structure of which depends on the format of the document. The WHERE clause is used to query the triplification. The CONSTRUCT clause is then used to create the required target RDF. Alternatively, a SELECT clause can be used to output query results rather than a graph, or an ASK clause can be used to determine the presence of matches to a query pattern.

The object of this study was to investigate the difficulties users experience with these two paradigms.

[1] Corresponding author. Email: paul.warren@open.ac.uk

Nielsen [6] defines usability as having five components: learnability, efficiency, memorability, errors and satisfaction. Our chief focus is on errors. However, many of our participants had little experience of YARRRML or SPARQL Anything, and to a considerable extent we were studying the learning experience. We also believe that observation of common errors gives insight into the conceptual difficulties which users have with these mapping paradigms. It also gives an insight into the intuitiveness of the tools, since the difference between what the user intuitively does, and what is required of the user, is a source of error.

We chose YARRRML as an example of the use of path descriptions because we believe it to represent the state-of-the-art from the viewpoint of usability of RML mappings generation; see Iglesias et al. [7] for a recent comparison of user-friendly serializations for creating RDF. Similarly, we believe that SPARQL Anything represents the state-of-the-art for the triplification approach. Our goal was to recommend:

i. rules and guidelines for users to create YARRRML and SPARQL Anything code;
ii. future developments to YARRRML and SPARQL Anything to improve usability;
iii. further areas of investigation and development for mapping techniques generally.

With regard to (i), we wished to investigate whether there are some use cases which are more appropriate for one or other of the two approaches.

Section 2 describes some related work on mapping tools. Section 3 gives an overview of the study and discusses the methodology used. Section 4 describes the questions and the data used. Sections 5 and 6 describe solutions to these questions for YARRRML and SPARQL Anything, and explain the two approaches. Sections 7 and 8 describe the participants' behaviours when using YARRRML and SPARQL Anything, in particular the mistakes they made. Section 9 compares the problems experienced with the two paradigms. Section 10 discusses the limitations of the study. Section 11 makes some recommendations, addressing (i) and (ii) above. Finally, Section 12 draws some general conclusions and addresses (iii), including presenting some research questions for future investigation.

2. Related work

Many of the first attempts to map from structured formats to RDF worked with the relational model; [8] provides a comparison of some of these early approaches. [9] and [10] provide references to some more recent approaches for mapping to RDF. An example of an early approach is the use of the mapping language R2RML. In Section 1 we discussed how R2RML was developed into RML[2], with the inclusion of a number of other source data formats, e.g. JSON and XML, and then into the more user-friendly YARRRML. In the following subsections, we describe a number of approaches, chosen to illustrate three themes: the extension of SPARQL, automatic triplification, and usability. In the final subsection of this section, we discuss the qualitative, observational approach we have used, and compare this to approaches used by other usability researchers in the Semantic Web community and elsewhere.

2.1. Extending SPARQL

Triplify was an early development which can be seen as a forerunner of SPARQL-based approaches [10]. Triplify used SQL queries to create RDF triples from a relational database. An advantage of this approach is the widespread familiarity with SQL, just as the subsequent SPARQL-based approaches benefit from a widespread familiarity with SPARQL.

Tarql[3] (SPARQL for tables) uses SPARQL syntax to query CSV directly [12]. Where a table has a first row containing headers, the elements of this row are used as variable names, otherwise ?a, ?b etc are used. ASK, SELECT and CONSTRUCT can then be used to query the CSV. With CONSTRUCT, users can create a triplification consistent with a required ontology.

SML (Sparqlification Mapping Language) used the syntax of the SPARQL CONSTRUCT clause to define mappings from a relational database to RDF [13]. The variables used in the CONSTRUCT clause are themselves equated to expressions derived from the relational database tables. The claim is that the SML syntax is a more compact syntax than R2RML. An evaluation showed that participants less experienced in R2RML preferred SML, found it more readable, and took less time to undertake a number of mapping tasks [13].

Whereas SML was concerned with translation from relational databases, SPARQL-Generate extends SPARQL to map to RDF from a variety of formats, e.g. CSV, JSON, XML, and HTML. [14]

[2] RML has now been drafted as a potential specification [11].

[3] https://github.com/tarql/tarql/; see also http://tarql.github.io/ for documentation.

provides an introduction to SPARQL-Generate, including a comparison with other approaches, whilst [15] provides a more detailed, formal description. In summary, SPARQL-Generate replaces the CONSTRUCT clause with a GENERATE clause and contains an ITERATE clause to equate variables to data elements, using path statements.

2.2. Automatic triplification

All the approaches discussed in this paper are used to create RDF triples. In that sense, they are examples of triplification. However, in this subsection we are concerned with approaches which create triplifications automatically or semi-automatically.

[16] describes a semi-automatic system for triplifying Wikipedia tables. The system "mines" DBpedia for predicates. They report a precision of 52.2% and believe this could be greatly improved through machine-learning. [17] describes a semi-automatic system, StdTrip, for transforming database schemas and instances to RDF triples, with particular emphasis on reuse of existing vocabularies. The paper makes a comparison with Triplify (discussed in the last subsection) and claims that StdTrip offers more support to users during the conceptual modelling phase. [18] describes a more recent system, CSV2RDF, for converting CSV files to RDF. The system takes account of embedded metadata and includes a GUI interface which can be used for modifying that metadata. The results of an experimental study indicate that the method is approximately linear in time. [18] also includes a relatively comprehensive survey of related work. [19] discusses the challenges of using a mapping language, such as RML, to match tabular data to knowledge graphs such as DBpedia and Wikidata. These challenges are analyzed in the context of the SemTab challenge[4]. Generally, automatic knowledge graph construction requires an iterative approach, and this might need to be taken account of in the further development of mapping languages. As an alternative, [19] suggests the declarative descriptions of workflows; presumably these would be used to create the iteration.

An approach closer to SPARQL Anything is described in [20]. The system converts geographic information described in JSON using a library (JSON2RDF) to RDF. The goal here is not to produce an end triplification, "but rather to automatically produce some kind of RDF that can then be transformed into a useful form simply using SPARQL CONSTRUCT queries".

SPARQL Anything has a similar philosophy to [20], but works with a range of formats, e.g. CSV, JSON, XML and HTML. It creates a triplification which can then be queried with ASK, SELECT and CONSTRUCT queries. As with Tarql and [20], the CONSTRUCT query enables a triplification to be created consistent with a desired ontology. [21] demonstrates theoretically that the SPARQL Anything approach is applicable to any file format expressible in BNF syntax as well as any relational database. The paper also compares the usability and performance of SPARQL Anything to other approaches, finding that it is comparable to other state-of-the-art tools.

SPARQL Anything is similar to SPARQL-Generate in that they are both extensions of SPARQL. However, SPARQL-Generate does not create a default triplification, but is a path-based approach making use of mappings defined similarly to mappings in YARRRML, i.e. with path statements written in JSONPath, XPath etc.

Figure 1 summarizes the three approaches to tripification: wholly manual approaches; semi-automatic approaches targeted at a pre-defined knowledge graph; and two-phase approaches with an initial wholly automatic phase followed by a manual phase using a SPARQL CONSTRUCT query.

2.3. Usability

The observation that R2RML is not user-friendly was the motivation for YARRRML [3]. The same observation has also motivated a number of graphical approaches. One example of such an approach is Juma, a block paradigm language designed initially for representing R2RML mappings [9], and then extended to SML [10]. Juma reduces syntax errors because it only permits the connection of blocks that create a valid mapping.

[4] https://www.cs.ox.ac.uk/isg/challenges/sem-tab/

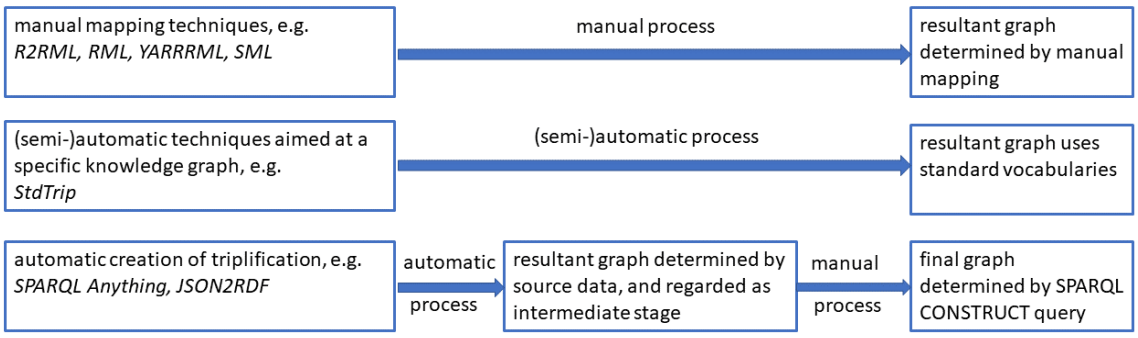

Fig. 1. three approaches to triplification: manual; (semi-)automatic; and automatic phase followed by a manual phase

There have been a few usability studies looking at mapping techniques. A study of Juma indicated that this approach could be used to create accurate mappings and that it achieved "good results in standard usability evaluations" [22]. [23] compared the mental workload associated with using R2RML and Juma, using two self-assessment techniques, Workload Profile [24] and NASA-TLX [25]. The conclusion was that there was little difference in mental workload but that Juma offered appreciably better performance.

[26] compared YARRRML, SPARQL-Generate, and ShExML; the last of these being a language based on Shapes Expressions (ShEx). The comparison required participants to map from a JSON file and an XML file onto an RDF graph, and used a combination of quantitative and qualitative methods. For the former, measurements included the time to perform a task, number of keystrokes and distance travelled by the mouse. For the latter, participants' feedback relating to usability was sought on a 5-point Likert scale. The study found that "ShExML users tend to perform better than those of YARRRML and SPARQL-Generate". More specifically, SPARQL-Generate was particularly difficult for first-time users. When comparing YARRRML and ShExML, it appeared that the superior performance of the latter was caused by details of syntax, e.g. "the use of keywords that made the language more self-explanatory and the modularity used on iterators which reminds of object-oriented programming languages". It is worth pointing out that the use-cases employed in [23] and [26] were considerably simpler than those used in our study, e.g. they did not have the kind of hierarchical data structures which we describe in Section 4.

We have focussed here on usability in the sense of Nielsen's [6] five components. There has also been work to understand the computational requirements of various approaches. [15] employed the simple use-case of mapping from CSV documents to RDF to compare the compute time requirements of RML and SPARQL-Generate. They found that SPARQL-Generate became faster for more than approximately 1,500 CSV rows, although observing that this will depend upon the implementation. They argued that, given the competitive performances, "ease of implementation and use is the key benefit of our approach [i.e. SPARQL-Generate]". In any case, compute-time equivalence for different approaches further strengthens the case for usability studies.

2.4. Studying user behaviour

The work cited in subsection 2.3 has largely used quantitative approaches, including Likert-style questionnaires, to study usability. [22] looked at accuracy, times to complete tasks, and also the results of post-task questionnaires assessing, e.g. system usefulness; although they also held informal post-task interviews. Whilst not an observational study in our sense, they do appear to have kept a note of the help participants required; the most commonly required help was "on how to interlink triples maps with the use of the parent triples map construct". [23] was a quantitative study using post-task questionnaires to assess mental workload. [26] used a post-task questionnaire, along with other measures, to assess acceptance of data integration languages.

Within the Semantic Web community, there appears to have been little use of qualitative, observational studies. One notable exception is provided by Pienta et al. [27], who used a think-aloud study to explore how participants reacted to novel features in their system for visually exploring graph query results. In software engineering research, observational studies have been used, e.g. to investigate how developers respond to problems [28]; whilst in HCI

observational studies are also used [29]. As Blandford [29] observes "people's ability to self-report facts accurately is limited". We believe observational studies complement both quantitative studies and self-report qualitative studies by providing insight into how users actually behave, rather than how they think they behave; and also insight into what they are thinking as they carry out tasks, rather than what they subsequently believe they thought.

3. Overview of the study and methodology

The study was a between-participants study with two conditions, i.e. one set of participants answered questions using YARRRML, the other set answered questions using SPARQL Anything. There were eight questions and these were the same in both conditions, in the sense that participants were presented with the same data files and with the same objectives. There were nine participants in the YARRRML condition, and nine in the SPARQL Anything condition. Participants were recruited from the Open University and from two W3C groups: the Knowledge Graph Construction Community [5] and the SPARQL 1.2 Community[6]. Participants were free to choose whether to work with SPARQL Anything or YARRRML.

Some days before the study, participants were provided with a tutorial which explained all they needed to know about the technique they were to use. They were also provided with a document which explained how to download the necessary software and contained the eight questions which they would be requested to answer during the study. They were also sent the data files, as discussed in Section 4, and the question files, as discussed in Sections 5 and 6[7]. They were requested not to look at these questions until the session. In recruiting participants, it was explained that the sessions would last one hour. At the beginning of each session, it was also explained that they might not answer all the questions in one hour, and that that was perfectly acceptable. Participants were also made aware that the study has been approved by the Open University's Human Research Ethics Committee (HREC/4195). Participants signed a consent form in which, apart from agreeing to take part in the study, they were also free to agree that their comments be quoted anonymously, or to withhold comments from publication. All agreed to their comments being quoted.

Before the study, participants were also sent a brief survey asking them about their previous experience with relevant technologies, and asking for some basic demographic information. Most participants were from Europe, with a few from the Americas and one from India. Ages varied from under thirty to over seventy, peaking at 40 to 49 years. There were 10 male and 8 female participants. At least six of the SPARQL Anything participants had a little, or more than a little, knowledge of SPARQL; only three had any knowledge of SPARQL Anything. Five of the YARRRML participants had a little, or more than a little, knowledge of RML or R2RML; only three had any knowledge of YARRRML. Table 5, in Section 10, lists the median knowledge of the two sets of participants in each of the relevant technologies. Eight of the participants classified themselves as software engineers; five as knowledge engineers; three as 'other'; and one did not specify role.

Each study was conducted over Microsoft Teams, with participants sharing their screen with the experimenter. There was a great deal of interaction between the participants and the experimenter; many participants found the exercises difficult and required assistance. This assistance ranged from 'hints', e.g. pointing out the presence of a square bracket denoting an array in JSON, to provision of the solution, which was then explained. Participants were also provided with files containing the required output RDF, although only a few participants referred to these. Only three of SPARQL Anything and four of the YARRRML participants completed all eight questions, although most completed the first five. Many participants spent more than the proposed hour on the study. Each session was recorded, using the Microsoft Teams recording facility and then analyzed using the NVivo qualitative analysis tool[8].

For reasons of time, and because we were chiefly interested in the conceptual, rather than the syntactic, difficulties experienced by the participants, we did not expect participants to create solutions from scratch. Instead, as is explained in more detail in Sections 5 and 6, we provided partial solutions and asked participants to complete the gaps.

[5] https://www.w3.org/community/kg-construct/

[6] https://www.w3.org/community/sparql-12/

[7] The tutorial, the question documents, and all the files sent out to participants are available at:
https://ordo.open.ac.uk/articles/online_resource/Materials_for_mapping_study_structured_data_to_RDF/21476883

[8] Supplied by QSR International: https://www.qsrinternational.com

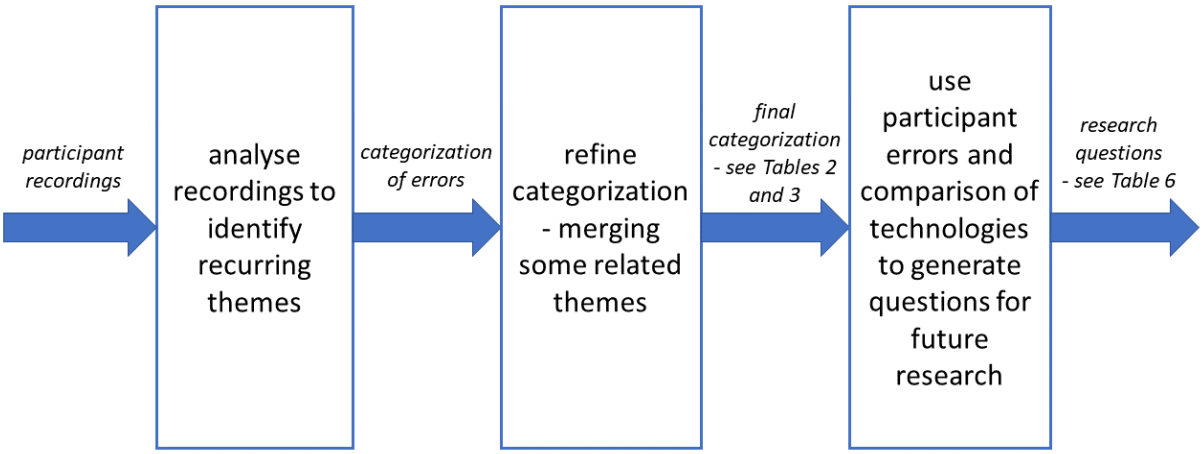

Fig. 2. Flowchart illustrating stages of analysis

We then analyzed the recordings to create a grounded classification of the observed errors, informed by the participants' comments[10]. This was inevitably a subjective process. We only make approximate attempts to quantify the importance of each error. In particular, an appreciable number of the participants found many of the questions very difficult and required considerable guidance, making rigorous quantification of errors impossible.

We initially coded the recordings to identify recurring categories of problems. As Norman observes [31], it is difficult to understand the participants mental models which are leading to these errors. However, we attempted to find categories which went beyond observed behaviours and reflected fundamental participant difficulties. After we had coded all the recordings, we reviewed the categories and merged some categories which reflected broadly similar underlying causes. This left us with six categories for YARRRML and five for SPARQL Anything. More detail is provided in Sections 7 and 8. Finally, we used these errors to generate research questions for further study, as described in Section 12. Figure 2 illustrates the overall process.

4. The questions and the data

Questions 1 and 2 used a slightly modified version of a JSON file which described an artwork in the Tate Gallery in London[11]. We chose a real, and relatively complex, example to ensure ecological validity. The file uses nested objects and arrays to provide information about the artwork, including characterizing the file with 13 topics. Each topic has a numeric id and a name. The topics are arranged hierarchically, in four levels. At the top-level there is one overarching topic, with id "1" and name "subject". We modified this file slightly: by re-ordering the information, to present the topic hierarchy top-down, and thereby improve readability; by renaming the overarching topic "topicRoot", to avoid any possibility of confusion with the word "subject" in the context of RDF triples or YARRRML mappings; by changing the JSON object "contributors" to "creator" and reducing the information provided about the creator. Figure 3 shows the resultant file. Note that "id" is reused three times: for the id of the creator (line 9), for the id of the artwork (line 13), and for the topic ids (lines 20, 24, etc.).

We omit the details of question 1, which was a straightforward question to test the basic understanding of the YARRRML or SPARQL Anything approaches, and to introduce participants to the study process. Question 2 used the JSON file in conjunction with the CSV file in Figure 4, which contains information about five artists. The goal of question 2 was to create one triple with subject the url of the

[10] For a description of grounded theory, see [30].

[11] The original JSON file is available at: https://github.com/tategallery/collection/blob/a51d8afc988ed083557e2950f4d0b644e7719f4a/artworks/a/000/a00002-1036.json

artwork, which is contained in the JSON file; and with object the url describing the artist, which is contained in the CSV file. The predicate of the triple was specified to be *dct:creator*. As with all the questions in the study, the predicate was provided in the question; only the subject and object were required to be identified from the data files. This requires correctly identifying the artist in the CSV file by matching the creator id from line 9 of the JSON file with the id in the first column of the CSV file.

Questions 3, 4 and 5 all had the same objectives. Question 3 used the JSON file shown in Figure 3. The objective was to create two sets of twelve triples. One set had as subject the *url* of the artwork, and as object an IRI of the form *tsub:id*, where each id was an id describing the artwork. The predicate was to be *schema:about*. We asked participants to exclude id = 1, since this topic is common to all artworks and carries no information. This did have the effect of making the questions more difficult. The other twelve triples required each *tsub:id* as subject, and the corresponding topic name as object, again excluding id = 1. The predicate was to be *schema:name*. Question 4 used an XML file created by the authors, and containing the same information as, and a similar structure to, the JSON file. The file, shown in Figure 5, was created without using attributes. Question 5 also used an XML file created by the authors, containing the same information as, and a similar structure to, the JSON file. This time, the file, shown in Figure 6, made maximum use of XML attributes. Note that, whereas *topics* (in the plural) is used once in the JSON and the previous XML files, as shown in Figures 3 and 5, in this XML file *topic* (in the singular) is used on multiple occasions. The order of questions 3, 4 and 5 was varied amongst the participants. Figure 7 shows the required output for the three questions.

Questions 6, 7 and 8 also had the same objective, and used the JSON file and the two XML variants. The objective of this question was to create 12 triples describing the links in the topic hierarchy, i.e. the subject of each triple was a topic, and the object was a child topic. The predicate was to be *skos:broader*. This time, id = 1 was included, so that *tsub:1* was the subject of two triples, with objects *tsub:29* and *tsub:91*. The order of presentation of these questions was also varied amongst the participants. Figure 8 shows the required output for these three questions. Table 1 illustrates the purpose of each of the questions, or sets of questions, and the implications for YARRRML and SPARQL Anything.

Table 1 Purpose of questions and required knowledge of YARRRML and SPARQL Anything

Q	Purpose of question	YARRRML requirements	SPARQL Anything requirements
1	Use the basic features of the two paradigms. Participants were required to generate an output file with triples created from a CSV and a JSON file.	Understand the structure of a mapping statement. However, there was no requirement to use a condition.	Create predicates of the form *xyz:<CSV column header>* and *xyz:<JSON name>*
2	Create a conditional join.	Use equal function, identifying which parameters come from the source and object mappings. See Figure 9.	Understand role of FILTER statement to create a join, and of IRI() function to create an IRI from a string. See Fig. 12.
3, 4, 5	Negotiate a hierarchical data structure, creating two sets of triples: one with items (ids and names) at the same level; the other associating a stated item (artwork) with all the topic ids in the hierarchy.	Use of JSONPath, XPath and recursive descent. See Figure 10.	Understand the difference between the triplification of JSON and XML. In particular, understand the different treatment of XML tags and attributes. Understand where one can use, e.g. *rdf:_1*, and where it is necessary to use a variable to bind to *rdf:_1*, *rdf:_2* etc. See Figure 13.
6, 7, 8	Negotiate a hierarchical data structure, linking each id with its immediate 'children' ids.	Similar to questions 3, 4, 5. However, care needs to be taken to link parent ids only to immediate children ids, i.e. not all children ids. See Figures 11 and 14.	

```
{
  "acno": "A00002",
  "acquisitionYear": 1922,
  "creator":
   {
    "name": "Robert Blake",
    "birthYear": 1762,
    "gender": "Male",
    "id": 38
   },
  "creditLine": "Presented by Mrs John Richmond 1922",
  "height": "213",
  "id": 1036,
  "medium": "Graphite on paper",
  "title": "Two Drawings of Frightened Figures, Probably for \u2018The Approach of Doom\u2019",
  "units": "mm",
  "url": "http://www.tate.org.uk/art/artworks/blake-two-drawings-of-frightened-figures-probably-for-the-approach-of-doom-a00002",
  "width": "311",
  "topics": {
   "id": 1,
   "name": "topicRoot",
   "children": [
    {
     "id": 29,
     "name": "emotions, concepts and ideas",
     "children": [
      {
       "id": 31,
       "name": "emotions and human qualities",
       "children": [
        {
         "id": 2815,
         "name": "fear"
        }
                    ]
      }
                ]
    },
    {
     "id": 91,
     "name": "people",
     "children": [
      {
       "id": 177,
       "name": "actions: expressive",
       "children": [
        {
         "id": 273,
         "name": "comforting"
        },
        {
         "id": 544,
         "name": "embracing"
        },
        {
         "id": 2653,
         "name": "recoiling"
        }
                    ]
      },
      {
       "id": 95,
       "name": "adults",
       "children": [
        {
         "id": 451,
         "name": "figure"
        }
                    ]
      },
      {
       "id": 97,
       "name": "groups",
       "children": [
        {
         "id": 799,
         "name": "group"
        }
                    ]
      }
                ]
    }
               ]
  }
}
```


Fig. 3. JSON file used in questions 1, 2, 3 and 6 (artwork.json)

1	id,name,gender,dates,yearOfBirth,yearOfDeath,placeOfBirth,placeOfDeath,url
2	37,"Blake, Benjamin",Male,c.1790–c.1830,1790,1830,,,http://www.tate.org.uk/art/artists/benjamin-blake-37
3	762,"Blake, John",Male,born 1945,1945,,"Rhode Island, United States",,http://www.tate.org.uk/art/artists/john-blake-762
4	763,"Blake, Peter",Male,born 1932,1932,,"Dartford, United Kingdom",,http://www.tate.org.uk/art/artists/peter-blake-763
5	38,"Blake, Robert",Male,1762–1787,1762,1787,"London, United Kingdom","London, United Kingdom", http://www.tate.org.uk/art/artists/robert-blake-38
6	39,"Blake, William",Male,1757–1827,1757,1827,"London, United Kingdom","London, United Kingdom", http://www.tate.org.uk/art/artists/william-blake-39

Fig. 4. CSV file used in question 2 (artist_data.csv)

1	<artwork>	44	<id>177</id>
2	<acno>A00002</acno>	45	<name>actions: expressive</name>
3	<acquisitionYear>1922</acquisitionYear>	46	<children>
4	<creator>	47	<item>
5	<name>Robert Blake</name>	48	<id>273</id>
6	<birthYear>1762</birthYear>	49	<name>comforting</name>
7	<gender>Male</gender>	50	</item>
8	<id>38</id>	51	<item>
9	</creator>	52	<id>544</id>
10	<creditLine>Presented by Mrs John Richmond 1992</creditLine>	53	<name>embracing</name>
11	<height>213</height>	54	</item>
12	<id>1036</id>	55	<item>
13	<medium>Graphite on paper</medium>	56	<id>2653</id>
14	<title>Two Drawings of Frightened Figures, Probably for 'The Approach of Doom'</title>	57	<name>recoiling</name>
15	<units>mm</units>	58	</item>
16	<url>http://www.tate.org.uk/art/artworks/blake-two-drawings-of-frightened-figures-probably-for-the-approach-of-doom-a00002</url>	59	</children>
17	<width>311</width>	60	</item>
18	<topics>	61	<item>
19	<item>	62	<id>95</id>
20	<id>1</id>	63	<name>adults</name>
21	<name>topicRoot</name>	64	<children>
22	<children>	65	<item>
23	<item>	66	<id>451</id>
24	<id>29</id>	67	<name>figure</name>
25	<name>emotions, concepts and ideas</name>	68	</item>
26	<children>	69	</children>
27	<item>	70	</item>
28	<id>31</id>	71	<item>
29	<name>emotions and human qualities</name>	72	<id>97</id>
30	<children>	73	<name>groups</name>
31	<item>	74	<children>
32	<id>2815</id>	75	<item>
33	<name>fear</name>	76	<id>799</id>
34	</item>	77	<name>group</name>
35	</children>	78	</item>
36	</item>	79	</children>
37	</children>	80	</item>
38	</item>	81	</children>
39	<item>	82	</item>
40	<id>91</id>	83	</children>
41	<name>people</name>	84	</item>
42	<children>	85	</topics>
43	<item>	86	</artwork>

Fig. 5. XML file used in questions 4 and 7 (artwork.xml)

```
<artwork acno = "A00002" acquisitionYear = "1922" height = "213" id = "1036" medium = "Graphite on paper" units ="mm" width ="311"
creditLine = "Presented by Mrs John Richmond 1922" title = "Two Drawings of Frightened Figures, Probably for 'The Approach of Doom'"
url ="http://www.tate.org.uk/art/artworks/blake-two-drawings-of-frightened-figures-probably-for-the-approach-of-doom-a00002">
  <creator name = "Robert Blake" birthYear = "1762" gender = "male" id = "38"/>
  <topic id = "1" name = "topicRoot">
    <topic id = "29" name = "emotions, concepts and ideas">
      <topic id = "31" name = "emotions and human qualities">
        <topic id = "2815" name = "fear"/>
      </topic>
    </topic>
    <topic id = "91" name = "people">
      <topic id = "177" name = "actions: expressive">
        <topic id = "273" name = "comforting"/>
        <topic id = "544" name = "embracing"/>
        <topic id = "2653" name = "recoiling"/>
      </topic>
      <topic id = "95" name = "adults">
        <topic id = "451" name = "figure"/>
      </topic>
      <topic id = "97" name = "groups">
        <topic id = "799" name = "group"/>
      </topic>
    </topic>
  </topic>
</artwork>
```

Fig. 6. XML file used in questions 5 and 8 (artworkAttributes.xml)

```
@prefix schema: <http://schema.org/> .
@prefix tsub:   <http://sparql.xyz/example/tate/topic/> .
@prefix xyz:    <http://sparql.xyz/facade-x/data/> .

tsub:273  schema:name  "comforting" .

tsub:177  schema:name  "actions: expressive" .

tsub:29  schema:name  "emotions, concepts and ideas" .

tsub:451  schema:name  "figure" .

tsub:2815  schema:name  "fear" .

tsub:97  schema:name  "groups" .

tsub:799  schema:name  "group" .

tsub:544  schema:name  "embracing" .

tsub:31  schema:name  "emotions and human qualities" .

tsub:95  schema:name  "adults" .

tsub:2653  schema:name  "recoiling" .

<http://www.tate.org.uk/art/artworks/blake-two-drawings-of-frightened-figures-probably-for-the-approach-of-doom-a00002>
schema:about  tsub:273 , tsub:95 , tsub:177 , tsub:29 , tsub:544 , tsub:2815 , tsub:31 , tsub:97 , tsub:2653 , tsub:91 , tsub:451 , tsub:799 .

tsub:91  schema:name  "people" .
```

Fig. 7. Required output for questions 3, 4 and 5

```
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix tsub: <http://sparql.xyz/example/tate/topic/> .
@prefix xyz:  <http://sparql.xyz/facade-x/data/> .

tsub:177  skos:broader  tsub:2653 , tsub:544 , tsub:273 .

tsub:29  skos:broader  tsub:31 .

tsub:97  skos:broader  tsub:799 .

tsub:31  skos:broader  tsub:2815 .

tsub:95  skos:broader  tsub:451 .

tsub:91  skos:broader  tsub:97 , tsub:95 , tsub:177 .

tsub:1  skos:broader  tsub:91 , tsub:29 .
```

Fig. 8. Required output for questions 6, 7 and 8

5. YARRRML – the solutions

This section presents solutions to the questions, using the questions to illustrate the features of YARRRML. As already noted, we omit the relatively straightforward question 1. A solution to question 2 is illustrated in Figure 9[12]. After the prefix statements, there is a mappings section, which contains *artworkMapping* beginning on line 5 and *artistMapping* beginning on line 19.

Both mappings start by specifying the source file, in lines 7 and 21. In each case, after the tilde, it is stated how the file should be interpreted, as JSON in the first case and as a CSV file in the second. At the end of line 7 is a dollar sign, in quotes. This is referred to as an iterator. It is a JSONPath expression, indicating over which parts of the JSON file the YARRRML should iterate to create the RDF. Specifically, the iterator should prepend the contents of any value statement in the mapping. Thus, in line 9 we have *url* in a value statement, to which is prepended *$*, from the iterator, to create the JSONPath *$.url*. Because the previous line contained *s:* we know that this JSONPath statement indicates the subject of the mapping. *po:* in line 10 indicates that the predicate and object will follow, and in this case, the predicate is *dct:creator*. Line 13 indicates that we wish to extract the object from another mapping, i.e. *artistMapping*, specifically from the file which is specified on line 21 as part of the definition of *artistMapping*. Because this is a CSV file, the path statement is simply a header from the first line of the file. We see from line 22 that the subject of *artistMapping*, and hence the object of the triple being formed, is indicated by the field name *url*. Note that line 22 begins with *s:* because it contains the subject (*url*) of *artistMapping*, although *url* is the object of the triple being formed. Because there are a number of records in the CSV file, we indicate the record to be used by the condition in lines 14 to 18. Specifically, a field *creator.id* in the JSON needs to be identical to a field *id* in the CSV file. In lines 17 and 18 the *s* and *o* before the closing square brackets indicate that, in the first case we are concerned with the mapping providing the source, i.e. *artworkMapping*, and hence the JSON file; and in the second case with the mapping providing the object, i.e. *artistMapping*, and hence the CSV file. *str1* and *str2* are dummy variables with no significance, e.g. they could be interchanged without changing the effect.

[12] In this figure, and in subsequent solutions to YARRRML and SPARQL Anything solutions, we show only those prefixes strictly necessary. In the study, for legacy reasons, files provided to participants contained some additional prefixes.

```
1  prefixes:
2    dct: http://purl.org/dc/terms/
3
4  mappings:
5    artworkMapping:
6      sources:
7        - ['artwork.json~jsonpath',"$"]
8      s:
9          value: $(url)
10     po:
11       - p: dct:creator
12         o:
13         - mapping: artistMapping
14           condition:
15             function: equal
16             parameters:
17               - [str1, "$(creator.id)",s]
18               - [str2, "$(id)",o]
19   artistMapping:
20     sources:
21       - ['artist_data.csv~csv']
22     s: $(url)
```

Fig. 9. YARRRML solution to question 2

Participants were not expected to create this YARRRML from scratch. We wanted participants to think conceptually about creating the mappings and be concerned as little as possible with the details of syntax. Consequently, participants were presented with a file as shown in Figure 9, except that the text in red italics was replaced with three dots. Participants were simply required to substitute a valid solution for these three dots. The missing elements, i.e. the elements represented by three dots, were chosen to test the participants' conceptual understanding of the mapping process; at the same time minimizing, as far as possible, the need to understand the details of the YARRRML syntax.

Figure 10 illustrates solutions to questions 3[13], 4 and 5. Again, the text in red italics was replaced by three dots. For each solution, there are two mappings, but this time they are independent, i.e. there is no condition connecting them. *artworkMapping* creates the twelve triples with subject the artwork url and with objects of the form *tsub:id*. *subjectMapping* creates the twelve triples with subjects of the form *tsub:id* and with objects the names of the topics.

Figure 11 shows solutions to question 6, 7 and 8. Question 6 requires two mappings. *topHierarchyMapping* creates the triples linking *topicRoot* to the two topics below it. *lowerHierarchyMapping* creates all the other required triples. Because of the structure of the XML data, questions 7 and 8 only require one mapping.

[13] The leading "topics" in line 14 should not be required. This was believed to be a problem with the RML mapper, see https://github.com/RMLio/rmlmapper-java/issues/150

1	prefixes:	prefixes:	prefixes:
2	tsub: "http://sparql.xyz/example/tate/topic/"	tsub: "http://sparql.xyz/example/tate/topic/"	tsub: "http://sparql.xyz/example/tate/topic/"
3	schema: "http://schema.org/"	schema: "http://schema.org/"	schema: "http://schema.org/"
4			
5	mappings:	mappings:	mappings:
6	artworkMapping:	artworkMapping:	artworkMapping:
7	sources:	sources:	sources:
8	- ['artwork.json~jsonpath',"*$*"]	- ['artwork.xml~xpath',"*/artwork*"]	- ['artwork.xml~xpath',"*/artwork*"]
9	s:	s:	s:
10	value: $(*url*)	value: $(*url*)	value: $(*@url*)
11	po:	po:	po:
12	- p: schema:about	- p: schema:about	- p: schema:about
13	o:	o:	o:
14	value: tsub:$(*topics..children[*].id*)	value: tsub:$(*//children/item/id*)	value: tsub:$(*topic//topic /@id*)
15	type:iri	type:iri	type:iri
16	subjectMapping:	subjectMapping:	subjectMapping:
17	sources:	sources:	sources:
18	- ['artwork.json~jsonpath', "*$..children[*]*"]	- ['artwork.xml~xpath', "//children/item"]	- ['artwork.xml~xpath', "//topic/topic"]
19	s:	s:	s:
20	value: tsub:$(*id*)	value: tsub:$(*id*)	value: tsub:$(*@id*)
21	po:	po:	po:
22	- p: schema:name	- p: schema:name	- p: schema:name
23	o:	o:	o:
24	value: $(*name*)	value: $(*name*)	value: $(@name)

Fig. 10. YARRRML solutions to (from left to right) questions 3, 4 and 5

1	prefixes:	prefixes:	prefixes:
2	tsub: "http://sparql.xyz/example/tate/topic/"	tsub: "http://sparql.xyz/example/tate/topic/"	tsub: "http://sparql.xyz/example/tate/topic/"
3	skos: "http://www.w3.org/2004/02/skos/core#"	skos: "http://www.w3.org/2004/02/skos/core#"	skos: "http://www.w3.org/2004/02/skos/core#"
4			
5	mappings:	mappings:	mappings:
6	lowerHierarchyMapping:	hierarchyMapping;	hierarchyMapping;
7	sources:	sources:	sources:
8	- ['artwork.json~jsonpath', "*$..children[*]*"]	- ['artwork.xml~xpath',"*//item*"]	- ['artworkAttributes.xml~xpath', "*//topic*"]
9	s:	s:	s:
10	value:tsub:$(*id*)	value: tsub:$(*id*)	value: tsub:$(*@id*)
11	po:	po:	po:
12	- p: skos:broader	- p: skos:broader	- p: skos:broader
13	o:	o	o
14	value:tsub:$(*children[*].id*)	value: tsub:$(*children/item/id*)	value: tsub:$(*topic/@id*)
15	type:iri	type: iri	type: iri
16	topHierarchyMapping		
17	sources:		
18	- ['artwork.json~jsonpath',"*$*"]		
19	s:		
20	value: tsub:$(*topics.id*)		
21	po:		
22	- p: skos:broader		
23	o:		
24	value: tsub:$(*topics.children[*].id*)		
25	type: iri		

Fig. 11. YARRRML solutions to (from left to right) questions 6, 7 and 8

6. SPARQL Anything – the solutions

As before, we omit question 1. Figure 12 illustrates a solution to question 2. Lines 9, 10, 13 and 14 were replaced by three dots in the question. As for YARRRML, the missing elements were chosen to test participants' conceptual understanding of the mapping process and minimize the need to understand details of syntax. To help participants, the terminator for the line was provided, either a semicolon (lines 9 and 13) or a full stop (lines 10 and 14). This practice was followed for all other questions. Participants were free to use the square bracket notation, as shown, or to create dummy variables or blank nodes.

To understand this solution, it is necessary to understand the triplification of CSV and JSON. For CSV, the document is regarded as a container, represented by a root node. This node is the subject of triples, with predicates *rdf:_1*, *rdf:_2* etc. The objects of these triples are nodes representing each of the rows, with the exception of the first row, which in our example is assumed to be a header row. Each of the column headers, e.g. *id* in Figure 4, is used to form a predicate with prefix *xyz:*, e.g. *xyz:id*. Each of the nodes representing rows is then the subject of triples with predicates of the form *xyz:id*, *xyz:name* etc. and with objects the literals in the cell elements[14].

For JSON, each object is regarded as a container and represented by a triple with a blank node as subject, and with predicate *xyz:name1*, where *name1* is the name of the JSON object. If the value is a literal, then this will be the object of the triple. If the value is another JSON object, with name *name2*, then the object of the triple will be another blank node, which in turn will be the subject of another triple, with predicate *xyz:name2*. If an object is an array, of size n, then the object of the triple will be a blank node, which in turn will be the subject of triples with predicates *rdf:_1*, *rdf:_2* … *rdf:_n*.

Figure 13 shows solutions to question 3, 4 and 5. These questions require the creation of a variable, here *?topicId*, to be included in the SERVICE clause and then used in line 14. In the solution to each question, lines 10 and 11 query the triplification. The solution to question 3 can be understood by referring to the JSON triplification explained above. To understand the solution to question 4, we need to understand how XML elements are triplified. Each distinct XML tag creates a node with IRI *xyz:tag*. Every element with that tag is regarded as a container and represented as a blank node, of type *xyz:tag*, i.e. the subject of a triple with predicate *rdf:type* and object *xyz:tag*. If the element only contains a literal, then the blank node will also be the subject of a triple with predicate *rdf:_1* and object the literal[15]. If the element contains n other elements, then the blank node will be the subject of triples with predicates *rdf:_1*, *rdf:_2* … *rdf:_n*, and objects blank nodes representing the n elements. As a result, the solution to question 4 follows the same pattern as question 3, except that lines 10 and 11 need to be changed. Here, *a* is a shorthand for *rdf:type* and *?li1*, *?li2*, *?li3* bind to *rdf:_1*, *rdf:_2* etc. Strictly speaking, in line 10, *a xyz:item* is not necessary, as all the elements directly below the *children* tag have tags *item*. Its use can be regarded as safeguarding against future development of the data. The solution to question 5 also follows the same form but needs to take account of the use of attributes in the XML. As before, when triplifying the XML, an element is represented by a blank node, of type *xyz:tag*. If the element contains an attribute then the blank node is also the subject of a triple, with predicate *xyz:name*, where name is the name of the attribute, and with object the value of the attribute. As before, lines 10 and 11 need to be changed. In line 10, the second *a xyz:topic* is not strictly necessary, since the only tags directly below *topic* are also *topic* tags. As in question 4, this can be regarded as safeguarding against future development of the data.

Figure 14 shows solutions to questions 6, 7 and 8. The differences between these three solutions are entirely limited to lines 9 and 10. For question 7, the *a xyz:item* in line 10 is not strictly necessary, as all the elements directly below the *children* tag have tag *item*.

Finally, for completeness although not relevant to the study, we note that SPARQL Anything treats XML CDATA sections as literals, and XML comments and processing instructions are ignored.

[14] For CSV files which do not have an initial header row, SPARQL Anything uses *rdf:_1*, *rdf:_2* etc to link the nodes representing each row with the cell elements.

[15] In the case where the element contains sub-elements and textual content (literal), then the predicate may be *rdf:_n*, where n is greater than 1, depending upon the position of the literal in the element. However, we did not consider this situation in our questions. In our questions, where a literal occurs in XML, it does not occur alongside sub-elements.

```
1  PREFIX xyz: <http://sparql.xyz/facade-x/data/>
2  PREFIX dct: <http://purl.org/dc/terms/>
3  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
4
5  CONSTRUCT {
6    ?artwork dct:creator ?artist .
7  } WHERE {
8    SERVICE <x-sparql-anything:location=artwork.json> {
9              [] xyz:url ?artworkUrl;
10               xyz:creator [ xyz:id ?creatorId ] .
11   }
12   SERVICE <x-sparql-anything:csv.headers=true,location=artist_data.csv> {
13           []  xyz:id ?artistId;
14               xyz:url ?artistUrl .
15   }
16   BIND (IRI(?artworkUrl) AS ?artwork) .
17   BIND (IRI(?artistUrl) AS ?artist) .
18   FILTER (xsd:integer(?artistId) = xsd:integer(?creatorId)) .
19 }
```

Fig. 12. SPARQL Anything solution to question 2

```
1  PREFIX xyz: <http://sparql.xyz/facade-x/data/>
2  PREFIX tsub: <http://sparql.xyz/example/tate/topic/>
3  PREFIX schema: <http://schema.org/>
4
5  CONSTRUCT {
6  ?artwork schema:about ?topic .
7  ?topic schema:name ?name .
8  } WHERE {
9    SERVICE <x-sparql-anything:location=artwork.json> {
10     [] xyz:children [ ?li [ xyz:id ?topicId; xyz:name ?name ]] .
11     [] xyz:url ?artworkUrl .
12   }
13   BIND(IRI(?artworkUrl) as ?artwork) .
14   BIND(IRI(CONCAT(STR(tsub:),STR(?topicId))) AS ?topic) .
15 }
```

```
1  PREFIX xyz: <http://sparql.xyz/facade-x/data/>
2  PREFIX tsub: <http://sparql.xyz/example/tate/topic/>
3  PREFIX schema: <http://schema.org/>
4
5  CONSTRUCT {
6  ?artwork schema:about ?topic .
7  ?topic schema:name ?name .
8  } WHERE {
9    SERVICE <x-sparql-anything:location=artwork.xml> {
10     [] a xyz:children; ?li1 [a xyz:item; ?li2 [ a xyz:id;
       rdf:_1 ?topicId]; ?li3 [ a xyz:name; rdf:_1 ?name ]] .
11     [] a xyz:url; rdf:_1 ?artworkUrl .
12   }
13   BIND(IRI(?artworkUrl) as ?artwork) .
14   BIND(IRI(CONCAT(STR(tsub:),STR(?topicId))) AS ?topic) .
15 }
```

```
1  PREFIX xyz: <http://sparql.xyz/facade-x/data/>
2  PREFIX tsub: <http://sparql.xyz/example/tate/topic/>
3  PREFIX schema: <http://schema.org/>
4
5  CONSTRUCT {
6  ?artwork schema:about ?topic .
7  ?topic schema:name ?name .
8  } WHERE {
9  SERVICE <x-sparql-anything:location=artworkAttributes.xml> {
10 [] a xyz:topic; ?li [a xyz:topic; xyz:id ?topicId; xyz:name ?name ] .
11 [] xyz:url ?artworkUrl .
12 }
13 BIND(IRI(?artworkUrl) as ?artwork) .
14 BIND(IRI(CONCAT(STR(tsub:),STR(?topicId))) AS ?topic) .
15 }
```

Fig. 13. SPARQL Anything solutions to questions 3 (top left), question 4 (top right), and question 5 (bottom)

```
1  PREFIX xyz: <http://sparql.xyz/facade-x/data/>
2  PREFIX tsub: <http://sparql.xyz/example/tate/topic/>
3  PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
4
5  CONSTRUCT {
6  ?parent skos:broader ?topic .
7  } WHERE {
8    SERVICE <x-sparql-anything:location=artwork.json> {
9      [] xyz:id ?parentId ;
10         xyz:children [ ?li [ xyz:id ?topicId ] ] .
11   }
12   BIND(IRI(CONCAT(STR(tsub:),STR(?parentId)))
     AS ?parent) .
13   BIND(IRI(CONCAT(STR(tsub:),STR(?topicId))) AS ?topic) .
14 }
```

```
1  PREFIX xyz: <http://sparql.xyz/facade-x/data/>
2  PREFIX tsub: <http://sparql.xyz/example/tate/topic/>
3  PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
4
5  CONSTRUCT {
6  ?parent skos:broader ?topic .
7  } WHERE {
8    SERVICE <x-sparql-anything:location=artwork.xml> {
9      [] a xyz:item; ?li1 [ a xyz:id; rdf:_1 ?parentId ] ;
10       ?li2 [ a xyz:children ; ?li3 [ a xyz:item ; ?li4 [ a xyz:id ;
         rdf:_1 ?topicId ] ] ] .
11   }
12   BIND(IRI(CONCAT(STR(tsub:),STR(?parentId)))
     AS ?parent) .
13   BIND(IRI(CONCAT(STR(tsub:),STR(?topicId))) AS ?topic) .
14 }
```

```
1  PREFIX xyz: <http://sparql.xyz/facade-x/data/>
2  PREFIX tsub: <http://sparql.xyz/example/tate/topic/>
3  PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
4
5  CONSTRUCT {
6  ?parent skos:broader ?topic .
7  } WHERE {
8    SERVICE <x-sparql-anything:location=artworkAttributes.xml> {
9      [] a xyz:topic; xyz:id ?parentId;
10       ?li1 [ xyz:id ?topicId ] .
11   }
12   BIND(IRI(CONCAT(STR(tsub:),STR(?parentId))) AS ?parent) .
13   BIND(IRI(CONCAT(STR(tsub:),STR(?topicId))) AS ?topic) .
14 }
```

Fig. 14. SPARQL Anything solutions to questions 6 (top left), 7 (top right), and 8 (bottom)

7. YARRRML – user behaviours

Table 2 shows the error categories, whilst the following subsections discuss these in more detail. Care needs to be taken in interpreting the numbers in the table. Because of the great deal of assistance given to some participants, these numbers should be interpreted as a lower bound.

With one exception, all or almost all of our participants made an error in each category. The exception was YARRRML syntax and semantics. As previously explained, the questions were constructed to reduce difficulties with using and interpreting YARRRML syntax; even so over half of our participants had difficulties.

7.1. Iterator errors

Fundamental to using YARRRML is understanding the role of the iterator, and the relationship between the iterator and value statements in a mapping. All our YARRRML participants had difficulties here, and made a number of mistakes.

Some users made the iterator path too long. For example, in question 3 (Figure 10, left-hand column), one participant correctly wrote *url* in the value statement in line 10, but wrote the iterator in line 8 as *$.children[*]*, rather than simply *$*. Concatenating the iterator and value statement then leads to *$.children[*].url*, which is clearly wrong. The converse problem is to make the iterator too short. In the first mapping (*artworkMapping*) of question 5 (Figure 10, right-hand column) a participant wrote / for the iterator and *@url* in the value statement on line 10. This omits *artwork*, i.e. the path to obtain the url, in full, is */artwork/@url*. Alternatively, there may be

no gap in the path statement obtained by concatenating iterator and value statement, but the iterator may still be too short. An example of this occurred in the second mapping (*subjectMapping*) of question 4 (Figure 10, middle column), where a participant wrote *//children* in the iterator and *item/id* and *item/name* in the value statements for the subject and object. The problem here is that, whilst the iterator and value statements concatenate to identify the topic id and name, there is no guarantee that id and name are correctly associated. In fact, the mapping creates 12 triples, but a number of the id's are repeated as subjects, and hence incorrectly associated with names. The iterator needs to extend sufficiently far down the hierarchy that the *id* and *name* are within the same *item*, i.e. the iterator should be *//children/item* and the value statements should be simply *id* and *name.*

Difficulties with the YARRRML iterator were, in fact, implied by [26], when comparing YARRRML and ShExML; see the comment in subsection 2.3 regarding the modular use of iterators in ShExML.

7.2. Recursive descent errors

Recursive descent was a cause of difficulty for nearly all of our participants. In *subjectMapping* of question 3, a participant wrongly assigned the recursive descent to the value statements, rather than the iterator. Thus, the iterator was written as *$.topics.children[*]* and the value statements contained *..id* and *..name*. This has a similar effect to the last example of the previous subsection, i.e. there is no guarantee that the correct *id* and *name* are associated. Another example of placing recursive descent in the value statements, instead of the iterator, occurred in question 6. In the mapping *lowerHierarchyMapping* (Figure 11, left-hand column), the participant wrote *$.topics* in the iterator, and *id* and *children[*]..id* in the subject and object value statements. This results in *lowerHierarchyMapping* creating twelve triples, linking *topicRoot* to each of the other topics. In fact, the iterator should be *$..children[*]* and the paths in the subject and object value statements should be *id* and *children[*].id*. In question 7, one participant put recursive descent into both the iterator and a value statement, writing *//item* for the iterator and *//children/item/id* in the second (object) value statement. This has the effect of creating 156 triples: *topicRoot* is linked to every other topic (12 triples) and all the topics below *topicRoot* are interlinked, in both directions, including to themselves (144 triples). It should be stressed that it is not necessarily wrong to put recursive descent into a value statement, as can be seen in the first mapping in the solution to questions 3, 4 and 5 (Figure 10).

Recursive descent was sometimes omitted completely, when it should have been used. In the value statement for the object of the first mapping of question 5, which should be written *topic//topic/@id*, a participant wrote *topic/topic/@id*, which has the effect of identifying only the two topics immediately below *topicRoot*. Note that the two occurrences of *topic* are necessary to avoid *topicRoot*. Another mistake was to write *topic/@id* for the object of the first mapping in question 5. This has the opposite effect to that required, i.e. it identifies only *topicRoot*. *//topic/@id* would also be wrong, since it would identify all the topics. At other times, participants may not have fully grasped the significance of recursive descent. In question 4, for the object of the first mapping, a participant wrote *topics/item//children/item/id*. This is not wrong, but *//children/item/id* would have been simpler.

7.3. Path syntax errors

There were also errors in the path syntax. With JSONPath, JSON arrays caused a problem for a number of participants. It may be that some participants simply failed to see the square brackets in the JSON. At least one participant omitted the asterisk from the square brackets in the JSONPath, i.e. wrote *[]* rather than *[*]* in question 3 (left-hand column of Figure 10). Another participant put the asterisk before the square brackets, i.e. writing *$..children*[]*. Other JSONPath errors were to write *.creator.id*, i.e. to use a leading dot, and to write *$topics*, i.e. to omit a dot. For XPath, there was a failure to use @ to identify an attribute, and also a failure to place a / before the @, i.e. writing *topic@id* rather than *topic/@id*. There was also confusion between JSONPath and XPath, e.g. between . and / as separators, and between $ and / to indicate the root of the document.

7.4. Path errors

More generally, nearly all our participants made errors in JSONPath and XPath. As an example of this, in question 2 a participant wrote *id* rather than *creator.id*, i.e. failing to differentiate between the three uses of *id*. One participant, in question 4, wrote *topics/children/item*, i.e. ignoring that *item* comes after *topics*; although in this case recursive descent

Table 2. YARRRML errors made by participants

Error	Examples
Iterator Errors (N = 9)	Iterator path too long, so that it cannot concatenate with path in a value statement to form correct JSONPath.
	Iterator path too short, so that there is a 'gap' when the iterator path is concatenated with a path in a value statement.
	Iterator path too short, and path in subject and object value statements too long, so that whilst they concatenate to form a valid JSONPath, the subject and object do not necessarily correspond.
	Not starting iterator from root of document, or with a recursive descent.
Recursive descent errors (N = 9)	Placing recursive descent in a value statement when it should be in the iterator. This can have a variety of effects, e.g. leading to no triples or too many. A variant on this is placing recursive descent in both the iterator and a value statement, when it should only be in the iterator.
	Omitting recursive descent when it is required, which can have the effect of creating only a subset of the required triples.
	Failing to realize that recursive descent can be used at the beginning of a path statement, to subsume the beginning of the path, rather than later in the statement. This causes unnecessarily long path statements.
Path syntax errors (N = 9)	Not taking account of JSON arrays.
	Misuse of dot in JSONPath: either including a leading dot (*.creator.id*) or omitting a dot (*$topics*).
	Failure to use @ to identify an attribute in XPath; alternatively, when @ was used, failure to place a / before the @, i.e. writing *topic@id* rather than *topic/@id*.
	Confusion between JSONPath and XPath, e.g. between . and / as separators and $ and / to indicate the document root.
Path errors, i.e. other than syntax errors (N = 8)	Path too short to uniquely specify data.
	Missing element in path.
Misunderstanding question or data (N = 8)	Failure to understand form of required triple.
	Copying JSON value rather than JSON name.
	Failure to understand nature of the data.
YARRRML syntax and semantics errors (N = 5)	Failure to understand purpose of mapping statement, i.e. that it needs to contain the name of a mapping.
	Failure to understand the significance of *s* and *o* in the join condition, and confusion about the role of the parameters *str1* and *str2*.
	Confusion between the use of quotes in the iterator but not in the value statement.
	Confusion potentially caused by use of $ in JSONPath and in YARRRML syntax.

should have been used. This problem may arise from not being fully aware of the structure of the data.

7.5. Misunderstanding question or data

Some participants failed to understand the form of the required triple, e.g. one participant thought that the subject of the *artistMapping* triple in question 2 (Figure 9) should be *id*, not *url*. Another problem was using a JSON value, rather than JSON name in the YARRRML. At least one participant failed to understand, or overlooked, the structure of the data, omitting the *item* tag between the *children* and *id* tags in a path statement in question 4 (Figure 10).

7.6. YARRRML syntax and semantics errors

Some participants had difficulty with the YARRRML syntax and semantics, i.e. the details of how YARRRML is used, as distinct from the more fundamental aspects of the mapping process such as the use of path statements and the relationship be-

tween iterator and value statement. This is consistent with the finding of [26], that superior performance with ShExML over YARRRML was caused by the details of syntax.

Question 1, which for brevity we have not described, required completing a mapping statement analogous to line 13 in question 2 (Figure 9). One participant did not understand that the name of a mapping was required, and instead looked in the relevant CSV file to find a seemingly appropriate field name. Interestingly, in the study of Juma, the block paradigm language for creating R2RML, the most common area where help was required was in interlinking mappings [22]. In question 2, the join condition caused problems. Some participants did not understand the significance of the *s* and *o*, to indicate subject and object, although this was explained in the tutorial. The use of *str1* and *str2* causes confusion. The YARRRML documentation explains that they are parameters for the equal function, and notes "that str1 and str2 can be switched as this does not influence the result of the equal function" [24, subsection 8.5]. One participant was confused by the fact that quotes are used around the JSONPath or XPath fragments in the iterator, but not in the value statement. The use of $ was also a source of confusion when manipulating JSON data. $ is used in JSONPath and also as part of the YARRRML syntax. In the latter case it is followed by a bracket and this led one participant to want to write a bracket after $ in JSONPath, e.g. *$(topics.children..)*. These last two points are related. In the iterator, the path fragment is in quotes. In the value statement, the path fragment is surrounded by brackets, with a leading dollar. Nevertheless, participants were, as far as possible, preserved from the difficulties of YARRRML syntax by the design of the questions. One participant commented "… there's so many little subtleties about, like, whether there needs to be a space here or not, but because you'd already filled in the YAML stuff, it was pretty straightforward".

8. SPARQL Anything – user behaviours

Table 3 shows the major error categories, whilst the following subsections discuss these in more detail. As with Table 2, care needs to be taken in interpreting the numbers in the table, because of the great deal of assistance given to some participants. Moreover, it is not always possible to be precise about the nature of the problem; in particular between problems in understanding the nature of the triplification and the structure of the data.

8.1. SPARQL errors

There were a number of errors which were syntactic, or at least related to SPARQL usage generally. Some participants had difficulty with the square bracket notation for blank nodes. For example, in question 6, one participant wrote *[] [xyz:id ?parentId; …]*. In fact, the tutorial examples illustrating SPARQL Anything used both this notation and explicit variables, e.g. *?b1*, *?b2* etc., and many participants chose the latter approach. There were difficulties, also, with variable names. In questions 3, 4 and 5, participants were required to create a variable to use in the SERVICE statement and in the BIND operator of line 14. One participant used *?topic*, which was pre-specified as the output of the BIND operator.

8.2. Graph pattern errors

There was difficulty in understanding how graph patterns mapped onto the triplification. In all but the first question, lines to be completed appeared as pairs of consecutive lines. It was generally understood that these lines should start with a blank node, but it was not always clear whether that should be a shared blank node. Figures 13 and 14, which show questions 3, 4, and 5; and questions 6, 7 and 8, illustrate that both cases can occur. For questions 3, 4, and 5, the second line to be completed (line 11) requires a starting blank node. For questions 6, 7 and 8, the second line (line 10) uses the initial blank node from the previous line. When confronted with the, rather complex, solution to question 7:

```
9   [] a xyz:item; ?li1 [ a xyz:id; rdf:_1 ?parentId ] ;
10          ?li2 [ a xyz:children ; ?li3 [ a xyz:item ; ?li4 [ a
xyz:id ; rdf:_1 ?topicId ] ] ]
```

one participant commented "but it seems like you connected magically …".

8.3. Misunderstanding the question or data

One problem in this category was in understanding the form of the required triple. However, many of the errors appeared to be caused by not fully appreciating the structure of the data. For example, in artwork.xml (Figure 5), overlooking that there is an *item* tag between each *children* and *id* tag; failing to discriminate between the creator *id* and the topic *id* in the description of the artwork; and in question 5 (Figure 13) not realizing the need for a new blank

node at the beginning of line 11. The last of these examples might also be interpreted as a problem with understanding the triplification.

8.4. Triplification errors

Fundamental to using SPARQL Anything is understanding the triplification; this was a source of difficulty for almost all of our participants. A great deal of assistance was given by the experimenter in explaining this. In particular, with nested data, the triplification can become confusing, and lead to errors. For example, in line 9 of question 7 (Figure 14, top right), one participant wrote:
?b1 a xyz:item; ?b ?parentId;

Instead of the line shown in the figure or, avoiding the square bracket notation:
?b1 a xyz:item; ?li ?b2 . ?b2 a xyz:id; rdf:_1 ?parentId; (1)

The problem here is that *id* is an XML element within *item* and it is the *id* object which contains *?parentId*. As written by the participant, the implication is that the *item* object directly contains *?parentId*.

Participants also needed to understand the two different ways which triplification is performed, i.e. that JSON names and XML attribute names are used to create predicates whilst XML element names are used to create a class. Moving between the two approaches is a source of confusion. For example, in question 4 (Figure 13, top right) a participant wrote: *[] xyz:url ?artworkUrl* where the correct solution should be, e.g. *[a xyz:url ; rdf:_1 ?artworkUrl]*, because question 4 used an XML file without attributes. The converse error occurred in question 5, i.e. writing the latter in place of the former, because question 5 used an XML file with attributes. Similarly in question 3, a participant started to write *[a xyz:children* , i.e. treating the JSON object name like an XML element name.

Another example of confusing the two triplification approaches occurred in question 7, where a participant wrote *[a xyz:topics ?parentId]* . In fact, starting with topics is starting too high up the hierarchy; however that is not the point we wish to make here. The participant did correctly recognize that *topics* is an element tag, and therefore should be converted into a class name. However, there was a failure to realize that *[a xyz:topics* is a complete triple pattern, and the participant may have slipped into thinking that *xyz:topics* was a predicate requiring to be followed by an object

Querying the triplification of XML poses a particular difficulty, when compared with the triplification of JSON. In statement (1) above, *?b1* and *?b2* are each the subject of two triples. In creating and reading this statement, having reached the first semi-colon, in order to recognize *?b1* as the subject of the next triple, the user needs to backtrack before continuing, or retain the *?b1* in working memory. A similar process occurs at the second semi-colon. This comment also applies when using the square bracket notation, as in the solution to question 7 shown in Figure 14, top right. However, the problem does not occur with the triplification of JSON, nor with the use of JSONPath and XPath in YARRRML.

8.5. Misuse of rdf:_1

More than half the participants wanted to use *rdf:_1* in graph patterns where they should have used a variable. SPARQL Anything uses the predicates *rdf:_1*, *rdf:_2* etc. in its triplification to reference the elements of JSON arrays. It also uses these predicates to reference XML elements nested within XML elements and to reference the literal contents of an XML element[16]. This means that use of these predicates is necessary in graph patterns to access a specific element of a JSON array, or an XML element or literal where its position is needed to uniquely specify it. Where we wish to access an XML element or literal which occurs alone within an outer XML element, we are free to use *rdf:_1* or a variable. However, *rdf:_1*, *rdf:_2* etc cannot be used in a pattern where we wish to range over the elements of a JSON array or over sibling XML elements. In these cases participants needed to use a variable which will bind to *rdf:_1*, *rdf:_2*, *rdf:_3* …, as appropriate. For example, in question 4 one participant wrote *[] a xyz:children ; rdf:_1 ?item1* in place of *[] a xyz:children ; ?li ?item1* . An alternative to using *?li* is to use the SPARQL Anything magic property *fx:anySlot*. However, for brevity, this possibility was not explained in the study.

[16] As explained in the footnote to Section 6, in our questions XML literals were always preceded by the predicate *rdf:_1*. This was because none of our questions used a composite model in which textual content and sub-elements were combined within the same XML element.

Table 3. SPARQL Anything errors made by participants

Error	Examples
SPARQL errors (N = 8)	Difficulties with square bracket notation, e.g. writing *[] [...*
	Difficulties with variable names, e.g. failure to understand need to create a new variable to be argument of *STR()* in questions 3, 4 and 5.
Graph pattern errors (N = 7)	Failing to understand when a blank node should be shared between graph patterns.
	Omitting an XML element or JSON object or array element in the path to the required datum.
	Ignoring JSON arrays.
	Failing to understand that a graph pattern can start from anywhere, not necessarily the root of the document.
Misunderstanding question or data (N = 7)	Failure to understand form of required triple.
	Failure to understand the data, e.g.: missing an element in an XML hierarchy, failure to discriminate between different uses of *id* in the data (creator id and topic id); or (in question 5) failure to understand the need for a new blank node on the second line to be completed.
Triplification errors (N = 6)	Failing to translate correctly from data to triplification, e.g. with nested data omitting a level.
	Confusing the two forms of triplification: JSON objects and XML attributes, versus XML elements.
Misuse of *rdf:_1* (N = 5)	Failing to note the need for a variable, rather than *rdf:_1*, where variable needs to bind to *rdf:_1* ... *rdf:_n*, as with JSON arrays or XML elements.

As with YARRRML, the requirement in questions 3, 4 and 5 to avoid *topicRoot* caused difficulty. In question 5, one participant wrote *[] a xyz:topic; xyz:id ?topicId* rather than *[] a xyz:topic; ?li [a xyz:topic; xyz:id ?topicId]* . In the latter, correct case, the initial triple pattern (*[] a xyz:topic*) is necessary to skip over the node which represents *topicRoot*.

Also as with YARRRML, JSON arrays were often ignored. This was a problem for more than half the participants, e.g. writing *[] xyz:children [xyz:id ?topicId]* instead of *[] xyz:children [?li [xyz:id ?topicId]]*, since SPARQL Anything creates a node to represent the array, and then nodes to represent each array element. Again as with YARRRML, data elements were sometimes overlooked, e.g. in question 7 missing out *item* which was intermediate between *children* and *id*. Moreover, the *id* of the artwork creator was not always identified uniquely, e.g. in question 2 one participant wrote *[] xyz:id ?artistId* in place of *[] xyz:creator [xyz:id ?artistId]*. The former will identify all the other uses of *id*, i.e. its use on line 13 to represent the id of the document, and all the uses of *id* representing the topic ids.

Finally, several participants started a graph pattern from the top of the document, where this was incorrect or unnecessary. In question 3, one participant wrote *?b1 xyz:topics ?b2 . ?b2 xyz:children ?b3 . ?b3 ?v ?b4 . ?b4 xyz:id ?id* . This only identifies the two topics immediately below *topicRoot*, because the use of *xyz:topics* binds *?b1* to the root of the document. A correct solution, as shown in Figure 13, starts from a blank node with name *xyz:children*. In this way, the graph pattern is able to match the RDF graph at all the appropriate levels of the hierarchy. One participant commented "I had no idea that we can start from anywhere … in the tree". Another participant commented "I'm stuck a bit in this mindset that I want to access something recursively … ideally like in an … XSLT way. I just wanted to like loop recursively through the whole tree …" .

9. Comparing the paradigms

The two paradigms are quite different, and require different expertise and understanding. The essential requirements to use YARRRML are familiarity with

path statements, e.g. JSONPath and XPath and to understand the role of the iterator. YARRRML is a subset of YAML, so some prior familiarity with YAML is useful. YARRRML is intended as a more human-readable alternative to RML. However, arguably it may be useful in debugging to at least be able to read RML. On the other hand, the essential requirements to use SPARQL Anything are familiarity with SPARQL syntax and semantics and to understand the particular triplification used. This means that some of the problems experienced by the two sets of participants seem quite different. However, in both cases the most fundamental problems have their roots in the same issues relating to the data. In particular, in our study, the need to negotiate hierarchical data created difficulties. For YARRRML, this meant that participants needed to understand how to use recursive descent. For SPARQL Anything, this meant that they had to understand how to construct a graph pattern that matched appropriately at various levels of the hierarchy.

How users find the advantages and disadvantages of the two paradigms will depend on their backgrounds. Those very familiar with JSONPath and XPath will have an advantage when starting to use YARRRML. Those very familiar with SPARQL will have an advantage when starting to use SPARQL Anything. However, an advantage of YARRRML over SPARQL Anything is that users of the former need only be familiar with the data and their required output RDF. Users of SPARQL Anything need also to be familiar with the triplification. On the other hand, once that triplification is understood, it is relatively easy for users to modify the SPARQL to make changes to the output, or simply to explore the data. Indeed, it is possible to explore the data without being fully aware of its structure, and of the triplification. One could, for instance, determine all the predicates in the triplification of a JSON file, thereby identifying all the names used in the file. Alternatively, with a triplification of XML, one could inspect the objects of all triples with predicates rdf:type to determine the tags used in the XML.

For YARRRML, the difference between JSON and XML resides in the difference between the JSONPath and XPath syntaxes. For SPARQL Anything, the difference is more fundamental, since XML tags are used to create class names. This not only creates a difficulty in moving between JSON and XML, but also makes for a greater complexity when dealing with XML, as we have noted in subsection 8.1.

It may be that YARRRML has an advantage where the required RDF is specified once and for all. Here an approach which avoids the intermediate stage of a triplification may be preferred. On the other hand, SPARQL Anything may have an advantage where we are unsure precisely what form the final output should take, or want to explore the data. In this case, the overhead of understanding the triplification will be worthwhile. We summarize our comparison in Table 4.

10. Limitations of the study

Before presenting our recommendations and conclusions, it is relevant to make some comments about the limitations of our approach.

For a qualitative study of this sort, we do not require as many participants as would be required in a quantitative study, where we are concerned with achieving statistical significance. When using Grounded Theory, Blandford proposes continuing until no new insights are being gained [29]. Although she suggests this typically occurs with between ten to twenty participants, we felt that, with nine participants per condition, towards the end of the study we were seeing the same repeated problems. A more serious problem was the nature of the participant sample. Obtaining participants for a study such as this is always difficult; participants need a relevant background in order to make sense of what is being asked of them. As a result, we were not able to balance the prior knowledge of our participants against the two paradigms. Table 5 shows the median prior knowledge of both sets of participants with regard to SPARQL, SPARQL Anything, RML or R2RML, and YARRRML.

Table 4. Comparison of YARRRML and SPARQL Anything

	YARRRML	SPARQL Anything
Users need expertise in …	Path statements, e.g. JSONPath and XPath; also some understanding of YAML. Ability to read RML may be useful in debugging.	SPARQL syntax and semantics.
Users need to understand …	The role of the iterator and how to use path statements.	The triplification.
	The use of recursive descent to negotiate the hierarchy.	How to construct a graph pattern to match at various levels of the hierarchy.
Advantages	Need only be familiar with data and required output, i.e. no intermediate stage.	Easy to change SPARQL to make changes to output or to explore the data.
Disadvantages	Not so easy to change required output and to explore the data.	Need to be familiar with triplification, which can be complex and differs for JSON and XML.
Recommended usage	Use cases where the output is fixed once and for all.	Use cases where ability to explore the data and flexibility in varying the output is required.

Table 5 Median prior knowledge
1 = no knowledge; 2 = a little knowledge;
3 = some knowledge; 4 = expert knowledge

	YARRRML participants	SPARQL Anything participants[†]
SPARQL	3	2.5
SPARQL Anything	1	1
RML or R2RML	2	1.5
YARRRML	1	1

[†]Based on eight of the nine participants; one participant did not provide the information.

Table 5 shows that neither of the two sets of participants had much knowledge of the two specific technologies under trial, i.e. SPARQL Anything and YARRRML. Nor did they have much knowledge of RML or R2RML. They did have rather more knowledge of SPARQL. In fact, even the YARRRML participants had more knowledge of SPARQL than of RML or R2RML. Ideally, it would have been good to include more people in both of the studies with more knowledge of RML or R2RML. In each group there were three participants who claimed some prior knowledge of the appropriate specific technologies, i.e. YARRRML or SPARQL Anything. These participants seemed to display the same mistakes as those with no knowledge; although the small number of participants make it impossible to draw any significant statistical inferences.

The fact that neither of the sets of participants had much prior knowledge of the specific technologies meant that we were studying the learning experience. This explains why our participants made a considerable number of errors. It gives us no indication of what kinds of errors might persist amongst experienced users. We can only conjecture that more superficial errors, e.g. syntactic errors, would diminish relatively rapidly with experience. Whereas the more conceptual errors, e.g. understanding the use of the iterator and recursive descent in YARRRML, and understanding the triplification of complex data structure in SPARQL Anything, are likely to diminish more slowly.

Our questions used the description of an artwork held by the Tate Gallery. We did this to achieve ecological validity; the JSON file was only a slight modification of a file used in a working application, whilst the two XML files were created from the JSON file. We have already pointed out, in subsection 2.3, that this resulted in a considerably more challenging study than the previous usability studies we have cited [23], [26]. However, we have no evidence that the data structures were representative of JSON and XML applications generally. To find a representative application would have necessitated a survey of JSON and XML-based applications. Our data structure was hierarchical, and inevitably a number of our participants' difficulties were con-

cerned with negotiating hierarchies. However, there might be other, quite different difficulties present in real-life applications which we were not able to examine.

11. Recommendations

In Section 1 we described three goals for our study: to recommend rules and guidelines for users of YARRRML and SPARQL Anything; to make recommendations for future developments of YARRRML and SPARQL Anything to improve usability; and to recommend areas of investigation and development for mapping techniques generally. In this section we discuss the first two of these goals; the next section will discuss the third goal.

11.1. Recommendations for users

Firstly, we present a set of recommendations which, if followed when writing YARRRML mappings, are likely to reduce many common errors. The first two are rules, which need to be followed. The second two are guidelines which will be helpful in many situations. These recommendations will also apply to other techniques which use an iterator and path statements.

1. The iterator path must start from the root of the document, or with a recursive descent.
2. The iterator path and each of the paths in corresponding value statements must concatenate to identify the required data element. In particular, there should be no overlap between iterator and value statement, and no gap between them.
3. Frequently, the iterator path should be as long as possible, and the corresponding value statements as short as possible, i.e. there should be no common elements at the start of the two value statements.
4. When dealing with hierarchical data, recursive descent may be necessary. Where the subject and object of the required triples vary over the hierarchy, the recursive descent is likely to be in the iterator. Where one of the subject or object is fixed, the recursive descent is likely to be in the value statement for the other.

For SPARQL Anything, we propose the following. The first two are rules, the final point is more properly described as a guideline.

1. JSON object names and XML attribute names must be used to create predicates in SPARQL triple patterns; whereas XML element tags must be used to create class names.
2. When writing SPARQL graph patterns, predicates *rdf:_1*, *rdf:_2* etc. must be used when we wish to access a JSON array element or an XML element or literal by position. Where we wish to iterate across an unknown number of subelements, a variable must be used to bind to *rdf:_1*, *rdf:_2* etc, or *fx:anySlot* should be used.
3. Graph patterns are not required to start from the root of a document. When dealing with hierarchical information, graph patterns may need to be designed to bind at various levels of the hierarchy.

11.2. Future developments for YARRRML and SPARQL Anything

Future developments need to reduce the possibility of the kinds of conceptual and syntactic errors we described in Tables 2 and 3.

YARRRML participants had difficulty in understanding the relationship between the iterator and corresponding value statements, with the result that the path in the iterator did not always properly concatenate with the paths in the value statement. These problems could be detected as the YARRRML is created, by comparing path statements with the structure of the data. Similarly, warning messages could be issued where there is commonality between two value statement paths, suggesting that this commonality might be moved into the iterator path. Going further, path evaluators to show the effect of path statements as they are being written, would aid users. These comments can also apply to other techniques based on path statements.

SPARQL Anything users had difficulty understanding the triplification. They face two problems: understanding how the data is triplified; and understanding how to query the triplification. One approach would be to automatically check that SPARQL Anything queries are consistent with the data, e.g. that the object of an *rdf:type* predicate is an IRI created from an XML tag, and not an XML attribute. Another approach would be to display part of the triplification, as required by the user, or to provide standard queries to interrogate the triplification.

Considering syntax, YARRRML is influenced very much by its historical legacy, and the fact that RML mappings are represented as RDF. The syntax contains features which appear more determined by

implementation than the requirements of defining the mappings. A prime example of this is the use of *str1* and *str2* in the condition, as in question 2. This is purely an implementation detail, and should be shielded from the user. The block paradigm offers one way of mitigating these problems. An alternative might be a tabular approach, with the user specifying the components of a mapping, e.g. source file, iterator, subject, predicate and object, in columns of a table. Whilst such a simple approach might not satisfy all requirements, it might satisfy the great majority of users. Another relatively simple approach has been proposed, based on an analogy with style sheets [32].

An issue with SPARQL is type conversion. Conversion to IRI is cumbersome, as can be seen from the solution to question 3. A simpler conversion mechanism would be useful in SPARQL generally, and particularly in the context of mapping to RDF. Since our study, SPARQL Anything has addressed this problem by the creation of a function *fx:entity* which casts its arguments to string, concatenates them and creates an IRI. Thus, line 14 in each of the solutions in Figure 13 could be rewritten:

BIND (fx:entity (tsub: , ?topicId) AS ?topic)

Similar changes could be made to lines 12 and 13 of the solutions in Figure 14.

Other additions to SPARQL Anything have been created to deal with sequences and container membership properties. One of these, *fx:anySlot*, has already been mentioned. In addition, there are a number of functions created to sequence through containers. The applicability of these functions has been demonstrated in the context of extracting musical features from musicXML files [33]. At the same time, [33] has identified the need for query modularisation to ease the design of sub-queries and avoid lengthy process pipelines; this is likely to be the topic of future research.

12. Conclusions and future directions

Our study compared two very different approaches for mapping data to RDF, using state-of-the-art examples of each. The differences between these two approaches are represented in the top and bottom process flows of Figure 1. At the top, the YARRRML user maps directly from the data to the desired RDF graph. This requires an understanding of the syntax and semantics both of YARRRML, including how to merge data from separate sources, and of the path statement language, e.g. JSONPath or XPath. At the bottom, the SPARQL Anything user is presented with an automatically created triplification, which is a lossless representation of the original data. The user needs to understand that triplification, and its relationship to the original data. The second part of the process is then achieved using SPARQL, with which the user of RDF is likely to be familiar.

Some problems are common to the two approaches. Most significantly, participants had difficulty with the hierarchical structures in our files. For YARRRML, this manifested itself in difficulties using recursive descent, with participants unclear about its use and whether to place recursive descent in the iterator or another path statement. For SPARQL Anything, the analogous problem was failing to understand that a graph pattern can start anywhere, not necessarily the root of the document, and bind at a variety of levels within the hierarchy; thereby picking out data items at all levels. More trivially, both sets of participants had difficulties with JSON arrays, perhaps in part through not detecting them in the JSON.

However, many of the problems experienced were specific to the particular approaches. What they share is a need to thoroughly understand the data and the use of the underlying paradigms. For YARRRML, the use of individual path statements is relatively straightforward; the difficulty frequently lies in the relationship between the iterator and the subject and object path statements. In training users, the correct design of this relationship needs to be stressed, with examples of the common use cases. One way of viewing the iterator is as a mechanism to allow path statements to share a common beginning, and then 'fork'. For SPARQL Anything, the difficulty is understanding and querying the triplification. Again, in training, emphasis on the various use cases is important.

Our study used real data to achieve a degree of ecological validity. However, we lack a clear view of the needs of the majority of users of mapping tools, e.g. which data formats they are predominantly interested in and what kind of data structures they are working with. Studying actual users, e.g. via surveys or focus groups, would enable the usability of future tools to be designed for the common use cases; perhaps accepting that the minority of 'power users' would require a greater degree of expertise to achieve their goals.

Our study was based on observing user behaviours, and in particular user errors. A future study would benefit from a more quantitative approach, e.g. con-

sidering the times participants take to respond and measuring cognitive load, e.g. using a tool such as NASA-TLX [25]. However, such a study would need to be based on sufficiently simple use cases that participants could complete without assistance. It would be useful, in such a study, to compare alternative triplifications. In particular, it would be valuable to consider an alternative triplification of XML which avoided creating classes from tags. It might be possible to use tags to create predicates, differentiating them from attribute names by the use of different namespaces.

Increased sophistication in the tools would aid users. An ideal would be tools which mirror the sophistication of a modern software development environment, in checking for 'compile-time' errors and making suggestions. We have made some recommendations in the context of YARRRML and SPARQL Anything. Where other tools are used, analogous compile-time features could be implemented.

A final question is whether there is an opportunity to bring together the two paradigms, incorporating the best features of each. One participant admitted to wanting to use JSONPath in SPARQL Anything, specifically to write *[] xyz:creator.id ?creatorId* rather than *[] xyz:creator [xyz:id ?creatorId]*, although knowing it was wrong. This suggests a more radical approach in which users could directly query the data with path statements, rather than query a triplification. Such an approach would need to extend the syntax of path statements, to achieve an effect equivalent to that of the use of the iterator in YARRRML. The work reported in [34] is relevant here; the paper defines a "lightweight query language" to navigate through JSON. This opens up the possibility of incorporating such a JSON query language within SPARQL, and similarly incorporating XQUERY to query XML.

In Table 6 we present a list of the research questions which we propose for future study.

Table 6. Research questions for future study

1	Can we characterize the use of structured data, so as to create a sample of use-cases which represents real-world usage?
2	How do YARRRML and SPARQL Anything, and similar techniques, compare with regard time to complete tasks and cognitive load imposed?
3	Can we develop an alternative triplification for XML which imposes less cognitive load than the current triplification?
4	Can we develop effective training for YARRRML and SPARQL Anything, and related technologies, which takes account of the most common use cases and appreciably improve user performance?
5	Is there a trade-off, as suggested in Section 9, between the YARRRML and SPARQL Anything approaches, i.e. with the former being better suited for stable situations, and the latter being better suited for exploratory investigation?
6	Can we develop tools to support these mapping technologies, so as to improve user performance? What features should such tools include?
7	Can we include path statements within SPARQL, so as to permit querying of structed data generally, without the need to consider an intermediate triplification? Would users find this proposed approach cognitively simpler than having to understand an intermediate triplification?

Acknowledgements

The authors would like to thank all those who gave up their time to participate in this study. The research has received funding from the European Union's Horizon 2020 research and innovation programme through the project SPICE - Social Cohesion, Participation, and Inclusion through Cultural Engagement (Grant Agreement N. 870811, https://spice-h2020.eu), and the project Polifonia: a digital harmoniser of musical cultural heritage (Grant Agreement N. 101004746, https://polifonia-project.eu).

References

[1] S. Das, S. Sundara, and R. Cyganiak, 'R2RML: RDB to RDF Mapping Language', W3C, Sep. 2012. [Online]. Available: https://www.w3.org/TR/r2rml/

[2] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van de Walle, 'RML: a generic language for integrated RDF mappings of heterogeneous data', in *Proceedings of the 7th Workshop on Linked Data on the Web*, CEUR, 2014.

[3] P. Heyvaert, B. De Meester, A. Dimou, and R. Verborgh, 'Declarative rules for linked data generation at your fingertips!', in *European Semantic Web Conference*, Springer, 2018, pp. 213–217.

[4] 'Tutorial: generating Linked Data with YARRRML'. Accessed: Jan. 18, 2023. [Online]. Available: https://rml.io/yarrrml/tutorial/getting-started/

[5] E. Daga, L. Asprino, P. Mulholland, and A. Gangemi, 'Facade-X: An Opinionated Approach to SPARQL Anything', in *Proceedings of the 17th International Conference on Se-*

mantic Systems, SEMANTiCS 2021, M. Alam, P. Groth, V. de Boer, T. Pellegrini, and H. J. Pandit, Eds., Amsterdam: IOS Press, Sep. 2021, pp. 58–73. [Online]. Available: http://oro.open.ac.uk/78973/

[6] J. Nielsen, 'Usability 101: Introduction to Usability'. Accessed: Aug. 05, 2023. [Online]. Available: https://www.nngroup.com/articles/usability-101-introduction-to-usability/

[7] A. Iglesias-Molina, D. Chaves-Fraga, I. Dasoulas, and A. Dimou, 'Human-Friendly RDF Graph Construction: Which One Do You Chose?', in *Web Engineering*, vol. 13893, I. Garrigós, J. M. Murillo Rodríguez, and M. Wimmer, Eds., in Lecture Notes in Computer Science, vol. 13893. , Cham: Springer Nature Switzerland, 2023, pp. 262–277. doi: 10.1007/978-3-031-34444-2_19.

[8] M. Hert, G. Reif, and H. C. Gall, 'A comparison of RDB-to-RDF mapping languages', in *Proceedings of the 7th international conference on semantic systems*, 2011, pp. 25–32.

[9] A. Crotti, C. Debruyne, and D. O'Sullivan, 'Juma uplift: using a block metaphor for representing uplift mappings', in *2018 IEEE 12th International Conference on Semantic Computing (ICSC)*, IEEE, 2018, pp. 211–218.

[10] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller, 'Triplify: light-weight linked data publication from relational databases', in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 621–630.

[11] A. Dimou, M. Vander Sande, B. De Meester, P. Heyvaert, and T. Delva, 'RDF Mapping Language (RML)', Nov. 2022. Accessed: Jan. 23, 2023. [Online]. Available: https://rml.io/specs/rml/

[12] B. DuCharme, 'Converting CSV to RDF with Tarql'. Accessed: Nov. 10, 2022. [Online]. Available: https://www.bobdc.com/blog/tarql/

[13] C. Stadler, J. Unbehauen, P. Westphal, M. A. Sherif, and J. Lehmann, 'Simplified RDB2RDF mapping', in *LDOW@ WWW*, 2015.

[14] M. Lefrançois, A. Zimmermann, and N. Bakerally, 'Flexible RDF generation from RDF and heterogeneous data sources with SPARQL-Generate', in *European Knowledge Acquisition Workshop*, Springer, 2016, pp. 131–135.

[15] M. Lefrançois, A. Zimmermann, and N. Bakerally, 'A SPARQL extension for generating RDF from heterogeneous formats', in *European Semantic Web Conference*, Springer, 2017, pp. 35–50.

[16] E. Munoz, A. Hogan, and A. Mileo, 'Triplifying wikipedia's tables', *LD4IE ISWC*, 2013.

[17] P. E. R. Salas, K. K. Breitman, M. A. Casanova, and J. Viterbo, 'StdTrip: An a Priori Design Approach and Process for Publishing Open Government Data.', in *SBBD (Posters)*, Citeseer, 2010, pp. 6–1.

[18] H. Mahmud, H. Jahan, S. Rashad, N. Haider, and F. Hossain, 'CSV2RDF: Generating RDF data from CSV file using semantic web technologies', *J. Theor. Appl. Inf. Technol.*, vol. 96, no. 20, pp. 6889–6902, 2018.

[19] A. Dimou and D. Chaves-Fraga, 'Declarative Description of Knowledge Graphs Construction Automation: Status & Challenges', in *Third International Workshop on Knowledge Graph Construction*, 2022.

[20] B. Norton, R. Krummenacher, and A. Marte, 'Linked Open Services: Update on Implementations and Approaches to Service Composition', in *Future Internet Symposium*, 2010. [Online]. Available: http://ceur-ws.org/Vol-647/

[21] L. Asprino, E. Daga, P. Mulholland, and A. Gangemi, 'Knowledge Graph Construction with a façade: a unified method to access heterogeneous data sources on the Web', *ACM Trans. Internet Technol. TOIT*, vol. In Press.

[22] A. Crotti, C. Debruyne, and D. O'Sullivan, 'Using a Block Metaphor for Representing R2RML Mappings.', in *VOILA@ ISWC*, 2017, pp. 1–12.

[23] A. Crotti, C. Debruyne, L. Longo, and D. O'Sullivan, 'On the mental workload assessment of uplift mapping representations in linked data', in *International Symposium on Human Mental Workload: Models and Applications*, Springer, 2018, pp. 160–179.

[24] P. S. Tsang and V. L. Velazquez, 'Diagnosticity and multidimensional subjective workload ratings', *Ergonomics*, vol. 39, no. 3, pp. 358–381, 1996.

[25] S. G. Hart, 'NASA-task load index (NASA-TLX); 20 years later', in *Proceedings of the human factors and ergonomics society annual meeting*, Sage publications Sage CA: Los Angeles, CA, 2006, pp. 904–908.

[26] H. García-González, I. Boneva, S. Staworko, J. E. Labra-Gayo, and J. M. C. Lovelle, 'ShExML: improving the usability of heterogeneous data mapping languages for first-time users', *PeerJ Comput. Sci.*, vol. 6, p. e318, 2020.

[27] R. Pienta *et al.*, 'VIGOR: interactive visual exploration of graph query results', *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 1, pp. 215–225, 2017.

[28] T. Lopez, H. Sharp, M. Petre, and B. Nuseibeh, 'Bumps in the Code: Error Handling During Software Development', *IEEE Softw.*, vol. 38, no. 3, pp. 26–34, 2020.

[29] A. E. Blandford, 'Semi-structured qualitative studies', Interaction Design Foundation, 2013.

[30] J. M. Corbin and A. Strauss, 'Grounded theory research: Procedures, canons, and evaluative criteria', *Qual. Sociol.*, vol. 13, no. 1, pp. 3–21, 1990.

[31] D. A. Norman, 'Some observations on mental models', in *Mental models*, A. Stevens and D. Gentner, Eds., Psychology Press, 2014, pp. 7–14.

[32] H.-J. Rennau, 'semsheets', Feb. 23, 2022. Accessed: Oct. 14, 2022. [Online]. Available: https://lists.w3.org/Archives/Public/semantic-web/2022Feb/0104.html

[33] M. Ratta and E. Daga, 'Knowledge Graph Construction From MusicXML: An Empirical Investigation With SPARQL Anything', 2022.

[34] P. Bourhis, J. L. Reutter, and D. Vrgoč, 'JSON: Data model and query languages', *Inf. Syst.*, vol. 89, p. 101478, 2020.